

# A polynomial expansion line search for large-scale unconstrained minimization of smooth $L_2$ -regularized loss functions, with implementation in Apache Spark\*

Michael B Hynes<sup>†</sup>

Hans De Sterck<sup>‡</sup>

## Abstract

In large-scale unconstrained optimization algorithms such as limited memory BFGS (LBFGS), a common subproblem is a line search minimizing the loss function along a descent direction. Commonly used line searches iteratively find an approximate solution for which the Wolfe conditions are satisfied, typically requiring multiple function and gradient evaluations per line search, which is expensive in parallel due to communication requirements. In this paper we propose a new line search approach for cases where the loss function is analytic, as in least squares regression, logistic regression, or low rank matrix factorization. We approximate the loss function by a truncated Taylor polynomial, whose coefficients may be computed efficiently in parallel with less communication than evaluating the gradient, after which this polynomial may be minimized with high accuracy in a neighbourhood of the expansion point. The expansion may be repeated iteratively in a line search invocation until the expansion point and minimum are sufficiently accurate. Our Polynomial Expansion Line Search (PELS) was implemented in the Apache Spark framework and used to accelerate the training of a logistic regression model on binary classification datasets from the LIBSVM repository with LBFGS and the Nonlinear Conjugate Gradient (NCG) method. In large-scale numerical experiments in parallel on a 16-node cluster with 256 cores using the URL, KDD-A, and KDD-B datasets, the PELS approach produced significant convergence improvements compared to the use of classical Wolfe approximate line searches. For example, to reach the final training label prediction accuracies, LBFGS using PELS had speedup factors of 1.8–2 over LBFGS using a Wolfe approximate line search, measured by both the number of iterations and the time required, due to the better accuracy of step sizes computed in the line search. PELS has the potential to significantly accelerate widely-used parallel large-scale regression and factorization computations, and is applicable to important classes of continuous optimization problems with smooth loss functions.

## 1 Introduction.

In large-scale optimization and machine learning, a common problem is the minimization of a loss function  $\mathcal{L}(\mathbf{w})$  with parameter vector  $\mathbf{w} \in \mathbb{R}^m$  [1]. For a set of  $n$  observations  $\mathcal{R} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$  with  $\mathbf{x}_i \in \mathbb{R}^p$  and  $y_i \in \mathbb{R}$ ,  $\mathcal{L}(\mathbf{w})$  is expressed as the mean of the individual losses  $\{f(\mathbf{w}; \mathbf{x}_i, y_i)\}$  evaluated at each observation with

an additional regularization term  $\lambda R(\mathbf{w})$ :

$$(1.1) \quad \mathcal{L}(\mathbf{w}) = \lambda R(\mathbf{w}) + \frac{1}{n} \sum_{i=1}^n f(\mathbf{w}; \mathbf{x}_i, y_i).$$

Here,  $\lambda$  is a fitting parameter that tunes the magnitude of the regularization penalty relative to the losses [2]. Often the regularization is an  $L_2$  penalty,  $\frac{1}{2}\|\mathbf{w}\|_2^2$ , such that  $R(\mathbf{w})$  is smooth with respect to  $\mathbf{w}$ . The optimization problem for minimizing (1.1) is

$$(1.2) \quad \mathbf{w}^* = \arg \min_{\mathbf{w}} \mathcal{L}(\mathbf{w}),$$

for which any unconstrained optimization algorithm may be applied. The use of nonsmooth regularization such as  $L_1$  penalties that induce sparsity is also an active research topic, however  $L_2$  penalties are commonly used for many practical problems [2, 3] and improving algorithmic performance in this case remains of significant interest.

The present work is situated in the context of batch or minibatch optimization algorithms that solve (1.2) through iterative updates with line searches, such as Gradient Descent (GD), LBFGS, truncated Newton algorithms, or the Nonlinear Conjugate Gradient (NCG) method (for a description of these methods, see e.g. Nocedal and Wright [4]). These algorithms have an update for the approximate solution  $\mathbf{w}_k$  in the  $k$ th iteration as

$$(1.3) \quad \mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k$$

where  $\mathbf{p}_k \in \mathbb{R}^m$  is the search direction and the scalar  $\alpha_k$  is a *step size* such that  $\mathcal{L}(\mathbf{w}_{k+1})$  is either exactly or approximately minimized in a univariate line search along the ray  $\mathbf{w}_k + \alpha \mathbf{p}_k$  with  $\alpha > 0$ . Approximate solutions that satisfy the Wolfe conditions are sought in practice at the expense of poorer convergence, since an inexact line search cannot in general be expected to perform better than an exact line search [4]. Since many commonly used univariate line search algorithms require evaluating both  $\mathcal{L}$  and  $\nabla \mathcal{L}$  in each line search iteration, they can be an expensive component of batch optimization algorithms.

Our main idea in this paper is simple but powerful: when (1.1) is smooth (infinitely differentiable), we propose to approximate  $\mathcal{L}(\mathbf{w} + \alpha \mathbf{p})$  in a univariate line search by a low-degree polynomial expansion, and show that the coefficients of this polynomial may be computed in a single pass over the dataset with modest communication requirements, after which the polynomial approximation may be minimized with high accuracy. In each line search invocation, the expansion may

\*Supported in part by the National Sciences and Engineering Research Council through a Canada Graduate Scholarship

<sup>†</sup>Department of Applied Mathematics, University of Waterloo, Waterloo, Canada. Email: mbhynes@uwaterloo.ca

<sup>‡</sup>School of Mathematical Sciences, Monash University, Melbourne, Australia. Email: hans.desterck@monash.edu

be repeated iteratively until the expansion point and minimum are sufficiently accurate. The advantages of our Polynomial Expansion Line Search (PELS) stem from two main improvements. Firstly, the PELS technique can obtain more accurate minima when the high intrinsic potential accuracy of the polynomial expansion is realized, which may lead to significantly fewer iterations of the optimization method to reach a desired accuracy than with classical approximate line searches. Secondly, if multiple iterations are required in a line search, the PELS method is much more efficient in terms of parallel communication than the iterations in approximate line searches that seek to impose the Wolfe conditions and require evaluating  $\nabla \mathcal{L}$  in each line search iteration: aggregating the polynomial coefficients requires much less communication than aggregating the  $m$ -dimensional gradient vectors  $\{\nabla f(\mathbf{w}; \mathbf{x}_i, y_i)\}$ . Additionally, when (1.1) is polynomial, the PELS expansion can be made *exact* with a sufficiently large degree.

The contributions of this paper are organized as follows. In §2, the PELS algorithm is presented for smooth loss functions, and a detailed example is given for  $L_2$ -regularized logistic regression. In §3 we detail an implementation in the Apache Spark framework [5] for fault-tolerant distributed computing, where it has been used to accelerate the training of logistic regression models by GD, NCG, and LBFGS on several large binary classification datasets. Finally, §4 compares the performance of these algorithms in Apache Spark using both PELS and a standard approximate line search scheme that may use multiple evaluations of  $\mathcal{L}$  and  $\nabla \mathcal{L}$  per line search invocation.

## 2 Background & Theory.

**2.1 Line Search Formulation.** A univariate line search is the unconstrained minimization problem of determining a step size  $\alpha^*$  that minimizes  $\mathcal{L}(\mathbf{w})$  along a fixed descent direction  $\mathbf{p}$ , formulated as

$$(2.4) \quad \alpha^* = \arg \min_{\alpha > 0} \mathcal{L}(\mathbf{w} + \alpha \mathbf{p}) = \arg \min_{\alpha > 0} \phi(\alpha).$$

Since computing  $\mathcal{L}(\mathbf{w} + \alpha \mathbf{p})$  in (2.4) is an expensive operation that requires evaluating  $f(\mathbf{w} + \alpha \mathbf{p}; \mathbf{x}_i, y_i)$  for each observation  $(\mathbf{x}_i, y_i)$ , an approximate solution to (2.4) is often sought instead of an exact solution in order to reduce the number of function evaluations required in the line search. Approximate line searches compute a sequence of iterates  $\{\alpha_j\}_{j \geq 0}$  until convergence criteria are satisfied, which are normally the Strong Wolfe conditions [4]. These conditions are a set of two inequalities guaranteeing (i) sufficient decrease as

$$(2.5) \quad \phi(\alpha) \leq \phi(0) + \nu_1 \alpha \phi'(0),$$

and (ii) a curvature condition requiring

$$(2.6) \quad |\phi'(\alpha)| \leq \nu_2 |\phi'(0)|$$

---

### Algorithm 1: Cubic Interpolating WA Line Search

---

**Input:**  $\alpha_0 > 0$

**Output:**  $\alpha_j$  satisfying (2.5) and (2.6) for  $\phi(\alpha)$

```

1  $I_0 \leftarrow [0, \alpha_0]$ 
2  $j \leftarrow 0$ 
3 while  $\alpha_j$  does not satisfy (2.5) and (2.6) do
4    $[\alpha_l, \alpha_u] \leftarrow I_j$ 
5    $P_j(\alpha) \leftarrow$  interpolate  $\phi(\alpha_l), \phi(\alpha_u), \phi'(\alpha_l), \phi'(\alpha_u)$ 
6    $\alpha_{j+1} \leftarrow \arg \min_{\alpha} P_j(\alpha)$ 
7    $I_{j+1} \leftarrow$  update interval using  $\alpha_l, \alpha_u, \alpha_{j+1}$ 
8    $j \leftarrow j + 1$ 
9 return  $\alpha_j$ 
```

---

for  $0 < \nu_1 < \nu_2 < 1$  (where  $\nu_1 \approx 10^{-4}$  and  $\nu_2 \approx 0.9$  [4]).

There are a multitude of widely-used inexact univariate line search algorithms for satisfying (2.5) and (2.6) for general  $\mathcal{L}(\mathbf{w})$  (see e.g. [6, 7, 8, 9]), and we refer to an algorithm in this class as a *Wolfe approximate* (WA) line search. Most successful WA algorithms are variants of the following classic interpolation scheme, summarized in Alg. 1. In each  $j$ th iteration of the line search, an interval  $I_j = [\alpha_l, \alpha_u]$  containing  $\alpha^*$  is determined, and the next  $\alpha_{j+1}$  is generated by the minimization of an interpolated cubic polynomial  $P_j(\alpha)$  with control points  $\phi(\alpha_l), \phi(\alpha_u), \phi'(\alpha_l)$ , and  $\phi'(\alpha_u)$  (i.e.  $P_j(\alpha_l) = \phi(\alpha_l)$ ,  $P'_j(\alpha_l) = \phi'(\alpha_l)$ , etc). The next interval  $I_{j+1}$  is computed such that the endpoints are closer to  $\alpha^*$ , and methods such as bisection, secant, and dual interpolation/minimization can be used to shrink the interval. Each evaluation of  $\phi(\alpha)$  and  $\phi'(\alpha)$  requires an  $\mathcal{O}[n]$  pass over  $\mathcal{R}$ , and it is typical in publicly available codes to structure optimization routines with a function that computes both  $\phi(\alpha)$  and  $\phi'(\alpha)$  simultaneously by evaluating  $\mathcal{L}(\mathbf{w} + \alpha \mathbf{p})$  and  $\nabla \mathcal{L}(\mathbf{w} + \alpha \mathbf{p})$ , explicitly returning a scalar and an  $m$ -dimensional vector such that, if the current step size is accepted, the computed value of  $\nabla \mathcal{L}(\mathbf{w} + \alpha \mathbf{p})$  provides  $\nabla \mathcal{L}(\mathbf{w}_{k+1})$  for the next iteration. Note, however, that if the initial  $\alpha_0$  in Alg. 1 satisfies (2.5) and (2.6), it is accepted as the solution to (2.4) *without* constructing and minimizing  $P_0(\alpha)$ , regardless of the accuracy of  $\alpha_0$ .

**2.2 Polynomial Expansion Line Search.** Our goal with the PELS method is to solve (2.4) accurately with as few *distributed* operations as possible when both  $R(\mathbf{w})$  and  $f(\mathbf{w}; \mathbf{x}_i, y_i)$  are smooth with respect to  $\mathbf{w}$ . In this case, we consider the Taylor expansion of  $\phi(\alpha) = \mathcal{L}(\mathbf{w} + \alpha \mathbf{p})$  in terms of  $\alpha$ , which requires summing the Taylor expansions of each  $f(\mathbf{w}; \mathbf{x}_i, y_i)$  in (1.1) in addition to an expansion for  $R(\mathbf{w})$ . For an expansion about a step size  $\alpha_j$ , we have

$$\mathcal{L}(\mathbf{w} + \alpha \mathbf{p}) = Q_{\lambda}(\alpha)|_{\alpha_j} + \sum_{i=1}^n \sum_{\ell=0}^{\infty} b_{\ell,i} (\alpha - \alpha_j)^{\ell}$$

where  $Q_\lambda(\alpha)|_{\alpha_j}$  is a polynomial expansion of  $\lambda R(\mathbf{w})$ , and  $\{b_{\ell,i}\}_{\ell=0}^\infty$  are the coefficients in the expansion of  $f(\mathbf{w}; \mathbf{x}_i, y_i)$  that depend explicitly on  $(\mathbf{x}_i, y_i)$ . To make this representation amenable to a distributed setting, we reorder the summations and write the degree- $d$  approximation to  $\mathcal{L}(\mathbf{w} + \alpha \mathbf{p})$  as

$$(2.7) \quad W(\alpha; \mathbf{w}, \mathbf{p})|_{\alpha_j} = Q_\lambda(\alpha)|_{\alpha_j} + \sum_{\ell=0}^d c_\ell (\alpha - \alpha_j)^\ell,$$

which has a truncation error  $\varepsilon_{d+1}(\alpha)|_{\alpha_j}$  as

$$(2.8) \quad W(\alpha; \mathbf{w}, \mathbf{p})|_{\alpha_j} = \mathcal{L}(\mathbf{w} + \alpha \mathbf{p}) - \varepsilon_{d+1}(\alpha)|_{\alpha_j}.$$

This error is an  $\mathcal{O}[(\alpha - \alpha_j)^{d+1}]$  term and hence small for  $\alpha$  near  $\alpha_j$ . Each coefficient  $c_\ell$  in (2.7) contains a summation over the dataset as

$$(2.9) \quad c_\ell = \sum_{i=1}^n b_{\ell,i} = \sum_{i=1}^n F_\ell(\mathbf{r}, \mathbf{p}; \mathbf{x}_i, y_i),$$

where  $\mathbf{r} = \mathbf{w} + \alpha_j \mathbf{p}$ , and the functions  $\{F_\ell\}_{\ell=0}^d$  compute the coefficients for the  $\ell$ th terms for a single observation.

The PELS algorithm minimizes the number of distributed operations required to solve (2.4) by exploiting the following facts: (i) computing the coefficients in (2.9) is a *parallelizable* operation, and (ii) once the  $\{c_\ell\}$  are computed,  $W(\alpha)|_{\alpha_j}$  is a useful approximation in a neighbourhood of  $\alpha_j$ . The PELS method proceeds as follows. Starting from the iterate  $\alpha_j$ , a polynomial approximation  $W(\alpha)|_{\alpha_j}$  is constructed by computing the coefficients  $\{c_\ell\}$  in parallel, after which the subsequent iterate is determined as

$$(2.10) \quad \alpha_{j+1} = \arg \min_{\alpha > 0} W(\alpha; \mathbf{w}, \mathbf{p})|_{\alpha_j}.$$

Solving (2.10) is a subproblem that will generate further iterates in a minimization routine, however, since the coefficients of  $W(\alpha)|_{\alpha_j}$  are *fixed*, the minimization problem requires no further distributed operations. In addition, computing the first and second derivatives  $W'|_{\alpha_j}$  and  $W''|_{\alpha_j}$  are inexpensive  $\mathcal{O}[d+1]$  operations in a minimization routine when performed with Horner's rule. If the truncation error  $\varepsilon_{d+1}(\alpha_{j+1})|_{\alpha_j}$  in (2.8) is determined to be too large, then  $\alpha_{j+1}$  is likely a poor approximation to  $\alpha^*$ , and the process repeats iteratively:  $\alpha_{j+1}$  is chosen as the new expansion point, and the coefficients of  $W(\alpha)|_{\alpha_{j+1}}$  are computed. This general form of the PELS algorithm is summarized in Alg. 2, where the coefficients for  $W(\alpha)|_{\alpha_j}$  are denoted by the vector  $\mathbf{c}_j = [c_0, c_1, \dots, c_d]^T$ , and the procedure `CALC_COEFFS` will be described in §3.1. As a termination condition at step 7 of Alg. 2, we use an approximation to the fractional error in the value of  $W(\alpha_{j+1})|_{\alpha_j}$ . To estimate the term  $\varepsilon_{d+1}(\alpha_{j+1})|_{\alpha_j}$  in the fractional error, we note that the truncation error in the degree- $d$  Taylor polynomial is heuristically bounded above by the

---

**Algorithm 2:** Polynomial Expansion Line Search

---

**Input:**  $\mathbf{w} \in \mathbb{R}^m, \mathbf{p} \in \mathbb{R}^m, \alpha_0 > 0, \theta > 0$

**Output:**  $\alpha_j \approx \alpha^* = \arg \min_{\alpha} \mathcal{L}(\mathbf{w} + \alpha \mathbf{p})$

---

```

1  $j \leftarrow 0$ 
2 repeat
3    $\mathbf{c}_j \leftarrow \text{CALC\_COEFFS}(\mathbf{w}, \mathbf{p}, \alpha_j)$ 
4    $\alpha_{j+1} \leftarrow \arg \min_{\alpha} W(\alpha; \mathbf{w}, \mathbf{p})|_{\alpha_j}$ 
5    $\epsilon_{j+1} \leftarrow \text{approximate } \varepsilon_{d+1}(\alpha_{j+1})|_{\alpha_j}$ 
6    $j \leftarrow j + 1$ 
7 until  $\left| \frac{\epsilon_j}{W(\alpha_j; \mathbf{w}, \mathbf{p})|_{\alpha_{j-1}}} \right| \leq \theta$ 
8 return  $\alpha_j$ 
```

---

**Procedure** `CALC_COEFFS`( $\mathbf{w}, \mathbf{p}, \alpha_0$ )

---

```

1  $\mathbf{r} \leftarrow \mathbf{w} + \alpha_0 \mathbf{p}$ 
2 Broadcast  $\mathbf{r}, \mathbf{p}$  to  $n_p$  compute nodes
3 for compute node  $t \in \{1, \dots, n_p\}$  do
4   for  $\ell \in \{0, \dots, d\}$  do
5      $c_\ell^{[t]} \leftarrow \sum_{\text{local } i_t} F_\ell(\mathbf{r}, \mathbf{p}; \mathbf{x}_{i_t}, y_{i_t})$ 
6  $\mathbf{c}_\lambda \leftarrow \frac{\lambda}{2} [\|\mathbf{r}\|_2^2, 2\mathbf{r}^T \mathbf{p}, \|\mathbf{p}\|_2^2, 0, 0, \dots]^T$ 
7  $\mathbf{c} \leftarrow \mathbf{c}_\lambda + \bigoplus_{t=1}^{n_p} \mathbf{c}^{[t]}$ 
8 return  $\mathbf{c}$ 
```

---

error in the polynomial of degree  $d-1$ ; that is, in a relevant neighbourhood of the expansion point  $\alpha_j$ ,  $|\varepsilon_d(\alpha)| \approx |c_d(\alpha - \alpha_j)^d| \geq |\varepsilon_{d+1}(\alpha)|$ . Hence, we approximate  $\varepsilon_{d+1}(\alpha_{j+1})|_{\alpha_j}$  by  $c_d(\alpha_{j+1} - \alpha_j)^d$  at step 5 of Alg. 2.

When  $\mathcal{L}(\mathbf{w})$  is itself polynomial, such as for least squares regression and low-rank matrix factorization as formulated by e.g. Gemulla et al. [10], then  $\varepsilon_{d+1}(\alpha) = 0$  for sufficiently large  $d$ , and Alg. 2 can solve (2.4) *exactly* in a single step without further coefficient computations. For the more general case of analytic  $\mathcal{L}(\mathbf{w})$ , the advantages of Alg. 2 over Alg. 1 are twofold. Firstly, the communication costs are lesser in the distributed operations: the summation in (2.9) to compute the coefficients  $\{c_\ell\}$  requires communicating  $d+1$  *scalar* values for each  $(\mathbf{x}_i, y_i)$ , whereas the distributed computation of  $\nabla \mathcal{L}$  in Alg. 1 communicates the  $m$ -dimensional gradient vectors  $\{\nabla f(\mathbf{w}; \mathbf{x}_i, y_i)\}$ . The second—and most important—reason is that, unlike standard line searches that compute values for  $\mathcal{L}$  and  $\nabla \mathcal{L}$  for only a *single*  $\alpha_j$ , the distributed operation computing the  $\{c_\ell\}$  in PELS produces a model which is valid for an entire neighbourhood of  $\alpha_j$ ; thus, if  $\alpha_j$  is close to  $\alpha^*$ , the PELS method can compute a very accurate approximation to  $\alpha^*$  with only a single  $\mathcal{O}[n]$  pass over the dataset to evaluate the polynomial coefficients. This is illustrated in Fig. 1, where both Alg. 1 and Alg. 2 have been applied to the function  $\phi(\alpha) = \alpha e^\alpha + e^{-(\alpha-4)}$ . In Fig. 1a, iterates produced by Alg. 1 (implemented as in [6] with  $\nu_1 = 10^{-4}$  and  $\nu_2 = 0.9$ ) are shown, however only a single iterate

has been generated since (2.5) and (2.6) are satisfied at the input  $\alpha_0 = 1$ ; as such, the algorithm terminates with a relatively inaccurate minimum, rather than compute  $\nabla \mathcal{L}$  with an  $\mathcal{O}[n]$  pass over the data for another step  $\alpha_1$ . On the other hand, Fig. 1b shows PELS with a degree-3 polynomial approximation  $W|_{\alpha_0}$  to  $\phi(\alpha)$  for the same  $\alpha_0 = 1$ . Here,  $\alpha_1$  is determined as the solution to (2.10), and is visibly more accurate than  $\alpha_0$  in Fig. 1a (only  $d = 3$  is shown since  $W|_{\alpha_0}$  could not be distinguished from  $\phi(\alpha)$  at this scale for larger degrees). This example is not a toy problem, but a case that we have often observed experimentally for the LBFGS algorithm: though the initial step size  $\alpha_0 = 1$  is frequently accepted in Alg. 1,  $\alpha_0$  is an inaccurate solution to (2.4).

In our implementation of PELS, we used the NR method as the optimization routine for minimizing the polynomial  $W|_{\alpha_j}$  in step 4 of Alg. 2, which is equivalent to solving for the roots of  $W'|_{\alpha_j}$ . However, it is possible that  $W'|_{\alpha_j} < 0$  for  $\alpha > 0$  if  $\alpha_j$  is far from  $\alpha^*$  (i.e.  $W'|_{\alpha_j}$  has no relevant real roots). In this case, instead of using a minimization routine,  $\alpha_{j+1}$  was determined by NR iteration at  $\alpha_j$  as

$$(2.11) \quad \alpha^{NR} = \alpha_j - \frac{\phi'(\alpha_j)}{\phi''(\alpha_j)} = \alpha_j - \frac{c_1}{2c_2},$$

where only the coefficients  $c_1$  and  $c_2$  appear since  $\phi(\alpha_j) = W(\alpha_j)|_{\alpha_j}$ . Since in practice, NR iteration converged within machine precision to a stationary point of  $W|_{\alpha_j}$  in very few iterations, (2.11) was used as a default whenever the NR method starting from  $\alpha_j$  failed to compute a zero gradient within a tolerance of  $10^{-15}$  in fewer than 10 iterations. Equation (2.11) also indicates that very fast convergence is expected for PELS when  $\alpha_j$  is close to  $\alpha^*$ , since NR iteration is equivalent to using only a  $d = 2$  approximation, whereas the PELS method can utilize arbitrary  $d > 2$ . Finally, note that NR iteration for step 4 may not be convergent for some initial guesses. Naturally, other minimization algorithms (e.g. bisection or backtracking) may be applied to  $W|_{\alpha_j}$ , however empirically we have found that a simple step size selection method (described in §3.2) to generate guesses for  $\alpha_0$  in each invocation of PELS produces stable results.

**2.3 Logistic Regression.** Consider a logistic regression model with  $L_2$ -regularization for binary classification of each  $\mathbf{x}_i \in \mathbb{R}^m$  with a group label  $y_i \in \{0, 1\}$ . The loss function is derived from the maximum log likelihood of the probability  $\Pr[y_i = 0 | \mathbf{x}_i, \mathbf{w}]$  as

$$(2.12) \quad \mathcal{L}(\mathbf{w}) = \frac{\lambda}{2} \|\mathbf{w}\|_2^2 + \frac{1}{n} \sum_{i=1}^n \left( \log[e^{-\mathbf{w}^T \mathbf{x}_i} + 1] + \mathbb{N}(y_i) \mathbf{w}^T \mathbf{x}_i \right)$$

where  $\mathbb{N}(y_i) = 1 - \mathbb{I}(y_i)$  and  $\mathbb{I}(y_i)$  is a boolean indicator function equal to 1 only if  $y_i \neq 0$ . It can be shown

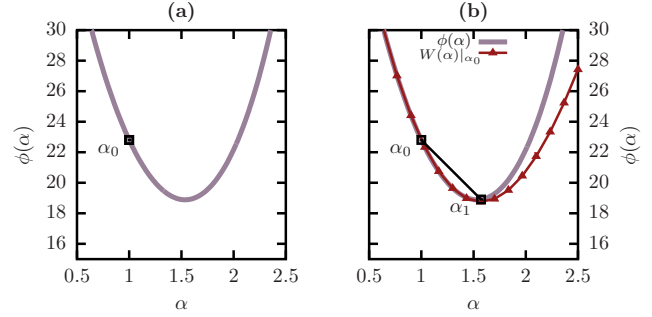


Figure 1: Example iterates  $\{\alpha_j\}$  produced by (a) Alg. 1 with  $\nu_1 = 10^{-4}$  and  $\nu_2 = 0.9$  (implementation from [6]) and (b) PELS with expansions using  $d = 3$  for  $\phi(\alpha) = \alpha e^\alpha + e^{-(\alpha-4)}$ .

that (2.12) is strictly convex and has a unique global minimizer [11]. Since (2.12) is infinitely differentiable, we may use a Taylor expansion about  $\alpha_0$  for  $\phi(\alpha) = \mathcal{L}(\mathbf{w} + \alpha \mathbf{p})$ . Denoting the ray  $\mathbf{w} + \alpha_0 \mathbf{p}$  by  $\mathbf{r}$ , and using the simplifications  $p_i = \mathbf{p}^T \mathbf{x}_i$  and  $r_i = e^{-\mathbf{r}^T \mathbf{x}_i}$ , the expansion up to 4th order is

$$(2.13) \quad \begin{aligned} \phi(\alpha) = & \frac{1}{n} \sum_{i=1}^n (\log[r_i + 1] + \mathbb{N}(y_i) \mathbf{r}^T \mathbf{x}_i) + \frac{\lambda}{2} \|\mathbf{r}\|_2^2 \\ & + (\alpha - \alpha_0) \left[ \frac{1}{n} \sum_{i=1}^n p_i \left( \mathbb{N}(y_i) - \frac{r_i}{r_i + 1} \right) + \lambda \mathbf{p}^T \mathbf{r} \right] \\ & + (\alpha - \alpha_0)^2 \left[ \frac{1}{n} \sum_{i=1}^n \frac{p_i^2 r_i}{2(r_i + 1)^2} + \frac{\lambda}{2} \|\mathbf{p}\|_2^2 \right] \\ & + \frac{(\alpha - \alpha_0)^3}{n} \sum_{i=1}^n \frac{p_i^3 r_i (r_i - 1)}{6(r_i + 1)^3} \\ & + \mathcal{O}[(\alpha - \alpha_0)^4]. \end{aligned}$$

The terms  $p_i$  and  $r_i$  in coefficients of (2.13) can be computed in parallel for each instance,  $(\mathbf{x}_i, y_i)$ . Furthermore, higher order coefficients in (2.13) depend only on the scalars  $p_i$  and  $r_i$ . Therefore, the only vector operations required in parallel are  $\mathbf{p}^T \mathbf{x}_i$  and  $\mathbf{r}^T \mathbf{x}_i$ .

**2.4 Apache Spark.** Apache Spark is a fault-tolerant, in-memory cluster computing framework designed to supersede MapReduce by maintaining program data in memory as much as possible between distributed operations. The Spark environment is built upon two components: a data abstraction, termed a resilient distributed dataset (RDD) [5], and the task scheduler, which uses a *delay scheduling* algorithm [12]. A Spark cluster is composed of a set of (slave) executor programs running on  $n_p$  compute nodes, and a master program running on a master node that is responsible for scheduling and allocating tasks to the compute nodes based on data locality. We describe the fundamental aspects of RDDs and the scheduler below.



RDDs are immutable, distributed datasets that are evaluated lazily via their provenance information—that is, their functional relation to other RDDs or datasets in stable storage. To describe an RDD, consider an immutable distributed dataset  $D$  of  $n$  records with homogeneous type:  $D = \bigcup_i^n d_i$  with  $d_i \in \mathcal{D}$ . The distribution of  $D$  across a computer network of  $n_p$  nodes  $\{v_t\}$ , such that  $d_i$  is stored in memory or on disk on node  $v_t$ , is termed its *partitioning* according to a partition function  $P(d_i) = v_t$ . If  $D$  is expressible as a finite sequence of deterministic operations on other datasets  $D_1, \dots, D_l$  that are either RDDs or persistent records, then its lineage may be written as a directed acyclic graph  $L$  formed with the parent datasets  $\{D_l\}$  as the vertices, and the operations along the edges. Thus, an RDD of type  $\mathcal{D}$  (written  $\text{RDD}[\mathcal{D}]$ ) is the tuple  $(D, P, L)$ .

Physically computing the records  $\{d_i\}$  of an RDD is termed its *materialization*, and is managed by the Spark scheduler program. To allocate computational tasks to the compute nodes, the scheduler traverses an RDD’s lineage graph  $L$  and divides the required operations into *stages* of local computations on parent RDD partitions. Suppose that  $R_0 = (\bigcup_i x_i, P_0, L_0)$  were an RDD of numeric type  $\text{RDD}[\mathbb{R}]$ , and let  $R_1 = (\bigcup_i y_i, P_1, L_1)$  be the RDD resulting from the application of a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  to each record of  $R_0$ . To compute  $\{y_i\}$ ,  $R_1$  has only a single parent in the graph  $L_1$ , and hence the set of tasks to perform is  $\{f(x_i)\}$ . This type of operation is termed a *map* operation, and has a *narrow* lineage dependency:  $P_1 = P_0$ , and the scheduler would allocate the task  $f(x_i)$  to a node that stores  $x_i$  since each  $y_i$  may be computed locally from  $x_i$ .

Stages consist only of local map operations, and are bounded by *shuffle* operations that require communication and data transfer between the compute nodes. For example, shuffling is necessary to perform *reduce* operations on RDDs, wherein a scalar value is produced from an associative binary operator applied to each element of the dataset. In implementation, a shuffle is conducted by writing the results of the tasks in the preceding stage,  $\{f(x_i)\}$ , to a local file buffer. These shuffle files may or may not be written to disk, depending on the operating system’s page table, and are fetched by remote nodes as needed in the subsequent stage via a TCP connection. Shuffle file fetches occur asynchronously, and multiple connections between compute nodes to transfer information are made in parallel. In addition, map tasks on the previous stage’s results that are stored locally by a compute node will occur concurrently with remote fetches.

A reduce operation on an RDD of type  $\text{RDD}[\mathcal{A}]$  produces a scalar value of type  $\mathcal{A}$  by an application of an associative binary operator  $\oplus : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$  to all of the elements  $\{a_i\}$  as  $\bigoplus_{i=1}^n a_i = a_1 \oplus a_2 \oplus \dots \oplus a_n$ . Reduce

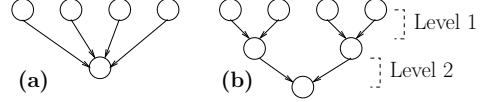


Figure 2: Comparison of an (a) all-to-one communication where every node, depicted by circles, communicates results with the driver (bottom circle), and a (b) multi-level scheme with  $n_l = 2$ , such that the final reduction to the driver requires communication with only 2 nodes.

operations require first performing *local* reductions on the partitions stored locally by each node before communicating the nodes’ results to the driver. The local reduction of the partition of  $\{a_i\}$  stored on node  $v_t$  is written  $a^{[t]}$  and computed as  $a^{[t]} = \bigoplus_{i \in \mathcal{I}_t} a_i$  for  $\mathcal{I}_t = \{i : P(a_i) = v_t\}$  and requires no communication between nodes. The full reduction for multiple nodes is then  $\bigoplus_{t=1}^{n_p} a^{[t]}$ , which incurs communication costs dependent on  $n_p$  and the size (in bytes) of an element in  $\mathcal{A}$ : reducing scalars ( $\mathcal{A} = \mathbb{R}$ ) is cheaper than reducing vectors ( $\mathcal{A} = \mathbb{R}^p$ ). Additionally, reduce operations on an RDD may be performed in two ways: either through an all-to-one communication pattern (Fig. 2a) in which all  $n_p$  local results from the compute nodes are communicated to the host machine on which the driver program is running, or through a multi-level tree communication [13] where intermediary (locally reduced) results are aggregated by compute nodes in  $n_l$  levels before being transferred to the driver [14] (Fig. 2b).

### 3 PELS Implementation & Performance Tests.

#### 3.1 PELS Implementation In Apache Spark.

The PELS method in Alg. 2 was implemented in Apache Spark based on the LBFSGS implementation in the Spark 1.5 codebase. The observations  $\mathcal{R} = \{(\mathbf{x}_i, y_i)\}$  were stored in an RDD with type  $\text{RDD}[(\mathbb{R}^p, \mathbb{R})]$ , where the vectors  $\{\mathbf{x}_i\}$  were either sparse or dense vectors with double precision. The parameter and search direction vectors  $\{\mathbf{w}_k\}$  and  $\{\mathbf{p}_k\}$  were not partitioned over the compute nodes, but stored as dense vector objects on the master node. Communicating the vectors  $\mathbf{p}_k$  and  $\mathbf{r}_k = \mathbf{w}_k + \alpha_j \mathbf{p}_k$  to the compute nodes in the cluster for any  $\alpha_j$  in the line search was performed via a torrent broadcast [14]. To compute the coefficient vector  $\mathbf{c}_j$  in any PELS iteration, the functions  $\{F_\ell(\mathbf{r}_k, \mathbf{p}_k; \mathbf{x}_i, y_i)\}_{\ell=0}^d$  in (2.9) were calculated as parallel map operations on the RDD of  $\mathcal{R}$ , and the resulting  $\mathbf{c}_j \in \mathbb{R}^{d+1}$  was summed via a multi-level reduce operation (as in Fig. 2b), where the operator  $\oplus$  was vector addition. The  $L_2$  regularization terms were computed by the master node as the vector  $\mathbf{c}_\lambda = \frac{\lambda}{2} [\|\mathbf{r}_k\|_2^2, 2\mathbf{r}_k^T \mathbf{p}_k, \|\mathbf{p}_k\|_2^2, 0, 0, \dots]$  (with zeros appended so  $\mathbf{c}_\lambda \in \mathbb{R}^{d+1}$ ) and added to  $\mathbf{c}_j$ . This is detailed in procedure COMPUTE\_COEFFS in Alg. 2, in which the local and global reductions occur at steps 5 and 7, respectively.

**3.2 PELS for LBFGS, NCG, & GD.** The PELS algorithm with  $d = 5$  was used to accelerate the following algorithms in Apache Spark for unconstrained optimization of smooth  $L_2$ -regularized loss functions: (i) LBFGS, (ii) NCG, and (iii) GD. The implementations using PELS will be henceforth denoted with a suffix *-P* as LBFGS-P, NCG-P, and GD-P. For LBFGS-P, NCG-P, and GD-P, once the search direction  $\mathbf{p}_k$  was determined at  $\mathbf{w}_k$  in the  $k$ th iteration, Alg. 2 was used to solve (2.4). For both the WA line search in LBFGS and PELS in LBFGS-P, the initial step size in each invocation (i.e. the input  $\alpha_0$  in Alg. 1 and Alg. 2) was taken to be 1, which is the recommended trial step for the LBFGS algorithm [15]. The NCG and GD algorithms with and without PELS both used a standard scaling formula [4] to determine the initial step size  $\alpha_k^0$  for the line search invocation in the  $k$ th iteration as

$$\alpha_k^0 = \alpha_{k-1} \frac{\nabla \mathcal{L}(\mathbf{w}_{k-1})^T \mathbf{p}_{k-1}}{\nabla \mathcal{L}(\mathbf{w}_k)^T \mathbf{p}_k},$$

where  $\alpha_{k-1}$  is the accepted step size from the previous iteration as in (1.3), and  $\alpha_k^0 = 1$  for  $k = 0$ . For NCG-P and GD-P, the search directions used to compute the coefficients in PELS and subsequently update  $\mathbf{w}_k$  were normalized as  $\mathbf{p}_k \leftarrow \mathbf{p}_k / \|\mathbf{p}_k\|_2$ . The NCG-P update rule was the positive Polak-Ribière formula [16], with a Powell restart condition [17]. For LBFGS-P, the initial inverse Hessian approximation in each iteration used Liu and Nocedal’s M3 scaling [15], as in the LBFGS code for Spark version 1.5.

**3.3 Algorithm Performance Tests.** To evaluate the efficacy of PELS, performance tests were conducted with a logistic regression model as in (2.12) that compared the LBFGS-P, NCG-P, and GD-P implementations with LBFGS, NCG, and GD using the WA cubic interpolating line search [7, 4] in the Breeze library [18] with  $\nu_1 = 10^{-4}$  and  $\nu_2 = 0.9$ , which is used in Spark’s LBFGS implementation. The LBFGS, NCG, and GD algorithms using a WA line search were implemented such that all vector operations other than the line search were identical to those in the LBFGS-P, NCG-P and GD-P implementations, respectively<sup>1</sup>. The Powell restart threshold in NCG-P was 0.2, however NCG had a restart threshold of 1.0 since smaller values often triggered restarts in *every* iteration, reverting the method to GD. For both LBFGS and LBFGS-P, a history of 5 corrections was used in our tests, as recommended for large problems [15].

The logistic regression models were trained on binary classification datasets procured from the LIBSVM

Table 1: *Properties of LIBSVM classification datasets used in numerical experiments, and respective magnitudes of  $\lambda$ .*

Dataset	$n$	$m$	$\text{nnz}(\mathbf{x}_i)$	$n_+ : n_-$	$\lambda$
Epsilon	400,000	2,001	2001	1.0 : 1.0	$10^{-8}$
RCV1 (test)	677,399	47,237	$74 \pm 54$	1.1 : 1.0	$10^{-7}$
URL	2,396,130	3,231,962	$117 \pm 17$	1.0 : 2.0	$10^{-8}$
KDD-A	8,407,752	20,216,831	$37 \pm 9$	5.8 : 1.0	$10^{-9}$
KDD-B	19,264,097	29,890,095	$29 \pm 8$	6.2 : 1.0	$10^{-9}$

repository [19], which are listed in in Tab. 1. This table also gives summary information about the datasets’ respective sizes of  $n$  and  $m$ , mean number of nonzero elements in  $\mathbf{x}_i$ , ratio of the number of true (nonzero)  $y_i$  labels to false  $y_i$  labels as  $n_+ : n_-$ , and magnitudes of  $\lambda$  used in (2.12). The dense  $\{\mathbf{x}_i\}$  in the Epsilon dataset were represented by contiguous dense vectors, while the other datasets’ instances were stored in compressed sparse vector format. All  $\mathbf{x}_i$  were further augmented as  $\mathbf{x}_i^T \leftarrow [\mathbf{x}_i^T \ 1]^T$  to implicitly include a constant offset term in the inner product  $\mathbf{w}^T \mathbf{x}_i$ . For all datasets, each algorithm was run with  $\mathbf{w}_0 = \mathbf{0}$ . In each iteration, computed values of  $\mathcal{L}(\mathbf{w}_k)$  and  $\|\nabla \mathcal{L}(\mathbf{w}_k)\|_2$  were written to the standard output filestream. The values of  $\theta$  used in Alg. 2 were  $\theta = 10^{-4}$  (approximately 0.01% error), except for the Epsilon and KDD-A datasets, on which  $10^{-6}$  was used. While smaller  $\theta$  generated more accurate step sizes, the additional PELS iterations increased the computational time; values of  $\theta \in [10^{-6}, 10^{-4}]$  were a good compromise between accuracy and speed.

All performance tests were performed on a computing cluster composed of: 16 homogeneous compute nodes, 1 storage node hosting a network filesystem, and 1 master node. The nodes were interconnected by a 10 Gb ethernet managed switch (PowerConnect 8164; Dell). Each compute node was a 64 bit rack server (PowerEdge R620; Dell) running Linux kernel 3.13 with two 8-core 2.60 GHz processors (Xeon E5-2670; Intel) and 200 GB of SDRAM. The master node had identical processor specifications and 512 GB of RAM. Compute nodes were equipped with six ext4-formatted 600 GB SCSI hard disks, each with 10,000 RPM nominal speed. The storage node (PowerEdge R720; Dell) contained two 6-core 2 GHz processors (Xeon E5-2620; Intel), 64 GB of memory, and a hard drive speed of 7,200 RPM.

Our Apache Spark assembly was built from a snapshot of the version 1.5 master branch using Oracle’s Java 7 distribution, Scala 2.10, and Hadoop 2.0.2. Input files to Spark programs were stored on the storage node in plain text. The compute nodes’ local SCSI drives were used for both Spark spilling directories and Java temporary files. Shuffle files were consolidated into larger files, as recommended for ext4 filesystems [20], and Kryo serialization was used. In our experiments, the Spark driver was executed on the master node

<sup>1</sup> We found that Spark’s LBFGS code had significant overheads stemming from the deeply nested data structures in Breeze; we wrote the LBFGS algorithm in an imperative style that took only ~65% as much time as the Spark code on large problems.

in standalone client mode, and a single instance of a Spark executor was created on each compute node. All RDDs had 256 partitions, corresponding to 1 partition per available physical core; performance decreased in general as more partitions were used. Finally,  $n_l$  was set to  $\log_2 16$  in all tree aggregations; empirically, this was faster for large datasets than the default of  $n_l = 2$ .

#### 4 Results & Discussion.

Our performance results are presented in two parts. In our initial tests, we are interested purely in the convergence improvements possible with PELS, and hence consider the Epsilon and RCV1 datasets that have little noise and are well-posed with  $m \ll n$ . In these tests, we present high-accuracy convergence traces since  $\mathcal{L}(\mathbf{w}^*)$  may be computed to machine precision. By contrast, real-life machine learning applications are often ill-posed and have statistical errors in the model or instances that exceed the numerical optimization error, such that  $\mathbf{w}^*$  is computed to only a few significant digits until the training loss ceases to decrease appreciably [21]. To demonstrate that PELS is effective in this practical setting as well, we present results on the large and ill-posed URL, KDD-A, and KDD-B datasets, and compute the acceleration in reaching terminal values of the training label prediction accuracy,  $\exp\{-\mathcal{L}(\mathbf{w}_k)\}$ , achievable by using PELS.

For the Epsilon and RCV1 datasets, respectively, Fig. 3 and Fig. 4 show the traces of  $|\mathcal{L}(\mathbf{w}_k) - \mathcal{L}(\mathbf{w}^*)|$  as a function of (a) iterations and (b) clock time for the algorithms considered in §3.3. In both plots,  $\mathcal{L}(\mathbf{w}^*)$  was determined as the minimal loss computed by any algorithm within the maximum number of iterations. That  $\mathbf{w}^*$  was computed accurately is evinced by the gradient norm at  $\mathbf{w}^*$ :  $\|\nabla \mathcal{L}(\mathbf{w}^*)\|_2$  was  $1.3 \times 10^{-11}$  for the Epsilon dataset and  $8.7 \times 10^{-12}$  for the RCV1 dataset, as computed by NCG-P. It is notable in both plots that NCG-P has drastically outperformed both the NCG and LBFGS algorithms that use a standard WA line search. LBFGS-P outperforms LBFGS in iterations and time for the Epsilon dataset, and performs similarly to LBFGS on the RCV1 dataset. GD-P and GD are virtually indistinguishable at the scale of Fig. 3 and Fig. 4 since there is little substantive difference in the two algorithms' traces.

The convergence traces for the large URL, KDD-A, and KDD-B datasets are shown in Fig. 5, Fig. 6, and Fig. 7, respectively (traces for GD and GD-P have been omitted from these plots since these algorithms made little progress towards the solution). These datasets required considerably more iterations and training time, and the LBFGS-P and NCG-P algorithms have outperformed their counterparts by multiple decimal digits in accuracy. However, since in many

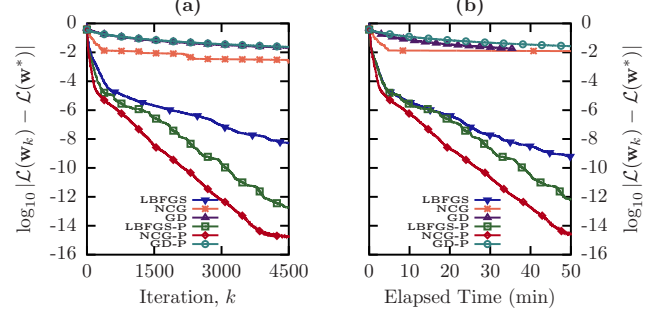


Figure 3: Convergence traces in regularized loss for the Epsilon dataset in (a) iterations and (b) elapsed time.

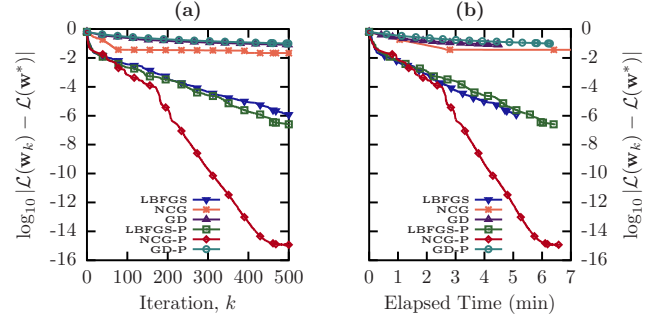


Figure 4: Convergence traces in regularized loss for the RCV1 dataset in (a) iterations and (b) elapsed time.

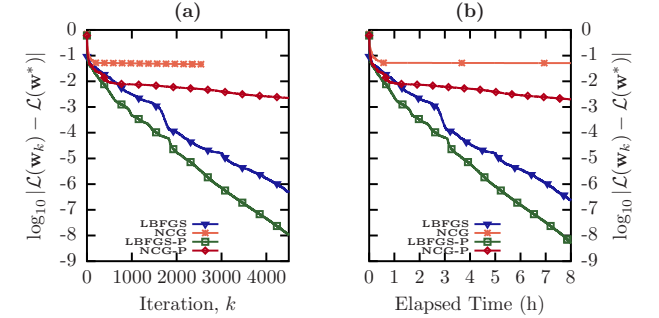


Figure 5: Convergence traces in regularized loss for the URL dataset in (a) iterations and (b) elapsed time.

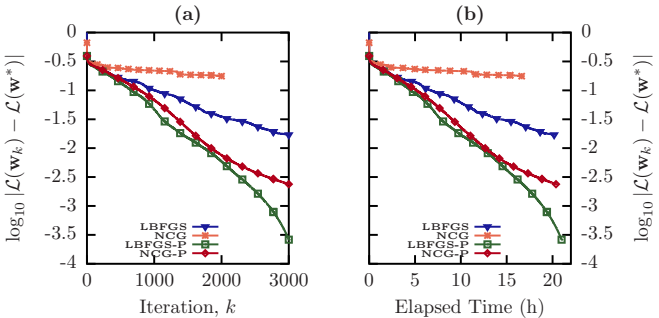


Figure 6: Convergence traces in regularized loss for the KDD-A dataset in (a) iterations and (b) elapsed time.

machine learning applications, the training procedure is halted once  $\exp\{-\mathcal{L}(\mathbf{w}_k)\}$  reaches a plateau, only  $\log_{10} |\mathcal{L}(\mathbf{w}_k) - \mathcal{L}(\mathbf{w}^*)| \approx -3$  may be necessary in practice. Bearing this, the speedup factors as a function of

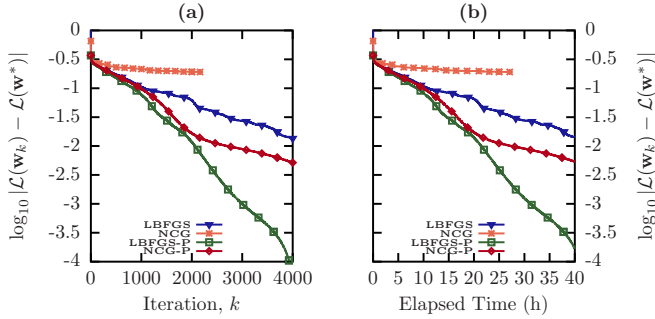


Figure 7: Convergence traces in regularized loss for the KDD-B dataset in (a) iterations and (b) elapsed time.

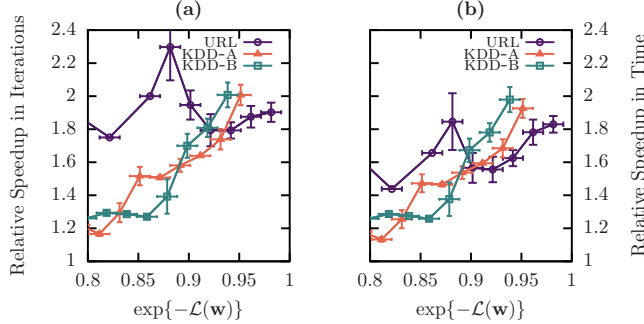


Figure 8: Speedup of LBFGS-P over LBFGS as a function of approximate training accuracy, computed for (a) iterations and (b) time.

training accuracy for LBFGS-P relative to LBFGS were computed for the URL and both KDD datasets, and are shown in Fig. 8, where the factors in Fig. 8a have been computed using the iterations required and the factors in Fig. 8b have been computed for clock time. These speedup factors were determined by finding the first iterate produced by LBFGS-P that reached the same or greater value of  $\exp\{-\mathcal{L}(\mathbf{w}_k)\}$  as LBFGS, and the error bars in this plot show the standard deviation about the mean speedup ratios computed in non-overlapping windows of width 0.02 along the x-axis. To reach the terminal training accuracies (e.g.  $\sim 97\%$  for URL), Fig. 8 shows factors of 1.8–2 in both iterations and clock time are achievable on the KDD-A, KDD-B, and URL problems.

To complement the convergence traces and explain the speedup in clock time of the PELS method in a distributed setting, Tab. 2 presents timing measurements for the distributed operations, averaged for each algorithm over all iterations. The quantity  $\tau_{fg}$  represents the mean clock time required to evaluate  $\mathcal{L}$  and  $\nabla\mathcal{L}$  through a map operation on the RDD of  $\mathcal{R}$  with a subsequent aggregation. However, since  $\mathcal{L}$  does not need to be computed explicitly in the PELS method,  $\tau_{fg}$  denotes the time required to compute *only*  $\nabla\mathcal{L}$  for LBFGS-P, NCG-P, and GD-P (although evaluating  $\nabla\mathcal{L}$  alone takes as much time as evaluating  $\mathcal{L}$  and  $\nabla\mathcal{L}$  simultaneously). For the algorithms using the PELS method,  $\tau_{ls}$  gives the mean time to both compute and

Table 2: Mean clock times for each iteration ( $\tau_{tot}$ ), evaluating  $\mathcal{L}$  and  $\nabla\mathcal{L}$  ( $\tau_{fg}$ ), and computing  $\mathbf{c}_j$  ( $\tau_{ls}$ ). The average number of function calls or line search iterations per outer iteration of the optimization algorithms are given by  $n_e$ .

	Alg.	$\tau_{tot}$ (s)	$\tau_{fg}$ (s)	$\tau_{ls}$ (s)	$n_e$
Epsilon	LBFGS	$0.5 \pm 0.2$	$0.42 \pm 0.07$		1.15
	NCG	$4 \pm 2$	$0.43 \pm 0.08$		9.92
	GD	$0.42 \pm 0.10$	$0.42 \pm 0.08$		1.01
	LBFGS-P	$0.7 \pm 0.1$	$0.42 \pm 0.07$	$0.32 \pm 0.06$	1.00
	NCG-P	$0.7 \pm 0.1$	$0.42 \pm 0.07$	$0.32 \pm 0.06$	1.00
	GD-P	$0.7 \pm 0.1$	$0.42 \pm 0.07$	$0.32 \pm 0.06$	1.00
RCV1	LBFGS	$0.6 \pm 0.3$	$0.47 \pm 0.08$		1.24
	NCG	$5 \pm 2$	$0.44 \pm 0.07$		10.48
	GD	$0.5 \pm 0.2$	$0.5 \pm 0.1$		1.06
	LBFGS-P	$0.8 \pm 0.2$	$0.45 \pm 0.07$	$0.29 \pm 0.06$	1.05
	NCG-P	$0.8 \pm 0.2$	$0.46 \pm 0.07$	$0.30 \pm 0.06$	1.08
	GD-P	$0.8 \pm 0.1$	$0.45 \pm 0.07$	$0.29 \pm 0.06$	1.00
URL	LBFGS	$6 \pm 2$	$4.5 \pm 0.2$		1.14
	NCG	$56 \pm 15$	$4.6 \pm 0.3$		11.10
	LBFGS-P	$6.1 \pm 0.3$	$4.3 \pm 0.2$	$1.0 \pm 0.1$	1.00
	NCG-P	$5.9 \pm 0.3$	$4.4 \pm 0.2$	$1.0 \pm 0.1$	1.01
KDD-A	LBFGS	$24 \pm 5$	$19 \pm 1$		1.05
	NCG	$30 \pm 18$	$19 \pm 1$		1.38
	LBFGS-P	$25 \pm 2$	$18 \pm 1$	$4.6 \pm 0.4$	1.03
	NCG-P	$24 \pm 2$	$18 \pm 1$	$4.7 \pm 0.6$	1.03
KDD-B	LBFGS	$37 \pm 8$	$28 \pm 2$		1.07
	NCG	$45 \pm 28$	$28 \pm 2$		1.41
	LBFGS-P	$38 \pm 2$	$27 \pm 2$	$6.7 \pm 0.6$	1.00
	NCG-P	$36 \pm 3$	$27 \pm 2$	$7.1 \pm 0.8$	1.00

aggregate the coefficient vector  $\mathbf{c}_j$ . The quantity  $n_e$  represents the average number of iterations per line search in the respective manners in which they were conducted: for the WA algorithms,  $n_e$  is the mean number of function/gradient evaluations per line search, and for the PELS algorithms, it represents the mean number of coefficient evaluations performed in Alg. 2 per line search. Thus, in each outer iteration of the optimization method, PELS algorithms perform 1 gradient computation taking  $\tau_{fg}$  seconds and then  $n_e$  operations lasting  $\tau_{ls}$  seconds, while WA algorithms perform  $n_e$  operations taking  $\tau_{fg}$  seconds. The total time per iteration is shown as  $\tau_{tot}$ . All uncertainty bounds show the standard deviation about the mean value; no uncertainty is given for  $n_e$  since it was taken as the ratio of the total number of line search calls to outer iterations.

The advantage of PELS for accurate large-scale distributed line searches is apparent when comparing  $\tau_{ls}$  to  $\tau_{fg}$  in Tab. 2, as well as the difference in  $n_e$  between PELS and WA algorithms. For all datasets,  $\tau_{ls} < \tau_{fg}$ , however when the problem size is large, such as for the KDD-A and KDD-B datasets, computing the PELS coefficients took only a quarter of the time required to compute  $\nabla\mathcal{L}$ . For all problems, LBFGS-P and NCG-P required a lower number of  $n_e$  evaluations in the line search than LBFGS and NCG, respectively, which entailed that the average  $\tau_{tot}$  was approximately the same for both PELS and WA implementations, despite the



fact that additional work was performed in computing the Taylor coefficients. In addition, note that  $n_e \approx 1$  for all PELS algorithms on the datasets considered, which is particularly effective when contrasted with NCG on the Epsilon, RCV1, and URL datasets: on these problems, NCG produced poorly scaled search directions requiring many line search iterations. While preconditioning in the NCG algorithm can be used to mitigate the poor scaling, it requires additional matrix-vector operations in each iteration, often constructing an approximation to the Hessian with LBFGS-type updates [22]; we thus find it notable that NCG-P often had better performance than LBFGS, *without* the need for preconditioning. In contrast to NCG, LBFGS generally had  $n_e \approx 1$  in Tab. 2; as such, the performance gains of LBFGS-P over LBFGS stem principally from reducing the total number of required iterations by computing more accurate minima in each line search invocation.

## 5 Conclusion.

In this paper, we have presented the Polynomial Expansion Line Search method for large-scale batch and mini-batch optimization algorithms, applicable to smooth loss functions with  $L_2$ -regularization such as least squares regression, logistic regression, and low-rank matrix factorization. The PELS method constructs a truncated Taylor polynomial expansion of the loss function that may be minimized quickly and accurately in a neighbourhood of the expansion point, and additionally has coefficients that may be evaluated in parallel with little communication overhead. Performance tests with our implementations of LBFGS, NCG, and GD with PELS in the Apache Spark framework were conducted with a logistic regression model on large classification datasets on a 16 node cluster with 256 processing cores. It was found, perhaps surprisingly, that NCG with PELS often exhibited better convergence and faster performance than LBFGS with a standard Wolfe approximate line search. For large datasets, the PELS technique also significantly reduced the number of iterations and time required by the LBFGS algorithm to reach high training accuracies by factors of 1.8–2. The PELS technique may be used accelerate parallel large-scale regression and matrix factorization computations, and is applicable to important classes of smooth optimization problems. All computer code for this paper is available through a `github` repository<sup>2</sup>.

## References

- [1] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2001.

<sup>2</sup><https://github.com/mbhynes>

- [2] G.-X. Yuan, C.-H. Ho, and C.-J. Lin, “Recent advances of large-scale linear classification,” *P IEEE*, vol. 100, no. 9, pp. 2584–2603, 2012.
- [3] G.-X. Yuan, K.-W. Chang, C.-J. Hsieh, and C.-J. Lin, “A comparison of optimization methods and software for large-scale  $L_1$ -regularized linear classification,” *J Mach Learn Res*, vol. 11, pp. 3183–3234, 2010.
- [4] J. Nocedal and S. Wright, *Numerical Optimization*. Springer Science & Business Media, 2006.
- [5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *P Conf Net Sys Des Impl*, pp. 15–39, USENIX, 2012.
- [6] J. J. Moré and D. J. Thuente, “Line search algorithms with guaranteed sufficient decrease,” *ACM T Math Software*, vol. 20, no. 3, pp. 286–307, 1994.
- [7] M. Al-Baali and R. Fletcher, “An efficient line search for nonlinear least squares,” *J Optimiz Theory App*, vol. 48, no. 3, pp. 359–377, 1986.
- [8] W. Hager, “A derivative-based bracketing scheme for univariate minimization and the conjugate gradient method,” *Comput Math Appl*, vol. 18, no. 9, pp. 779–795, 1989.
- [9] W. Hager and H. Zhang, “A new conjugate gradient method with guaranteed descent and an efficient line search,” *SIAM J Optimiz*, vol. 16, no. 1, pp. 170–192, 2005.
- [10] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, “Large-scale matrix factorization with distributed stochastic gradient descent,” in *P SIGKDD*, pp. 69–77, ACM, 2011.
- [11] C.-J. Lin, R. C. Weng, and S. S. Keerthi, “Trust region Newton method for logistic regression,” *J Mach Learn Res*, vol. 9, pp. 627–650, 2008.
- [12] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling,” in *P Euro Conf Comp Sys*, pp. 265–278, ACM, 2010.
- [13] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford, “A reliable effective terascale linear learning system,” *J Mach Learn Res*, vol. 15, no. 1, pp. 1111–1133, 2014.
- [14] B. Yavuz and X. Meng, “Performance improvements in MLlib.” <https://databricks.com/blog/2014/09/22/spark-1-1-ml-lib-performance-improvements.html>, 2014.
- [15] D. C. Liu and J. Nocedal, “On the limited memory BFGS method for large scale optimization,” *Math Program*, vol. 45, no. 1-3, pp. 503–528, 1989.
- [16] J. C. Gilbert and J. Nocedal, “Global convergence properties of conjugate gradient methods for optimization,” *SIAM J Optimiz*, vol. 2, no. 1, pp. 21–42, 1992.
- [17] M. J. Powell, “Restart procedures for the conjugate gradient method,” *Math Program*, vol. 12, no. 1, pp. 241–254, 1977.
- [18] D. Hall, “Breeze: numerical processing library for Scala.” <https://github.com/scalanlp/breeze>.
- [19] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Trans Intel Sys Tech*, vol. 2, no. 3, p. 27, 2011.
- [20] A. Davidson and A. Or, “Optimizing shuffle performance in Spark,” *UC Berkeley, Dept Elec Eng & Comp Sci*, 2013.
- [21] O. Bousquet and L. Bottou, “The tradeoffs of large scale learning,” in *Adv Neur In*, pp. 161–168, 2008.
- [22] W. W. Hager and H. Zhang, “A survey of nonlinear conjugate gradient methods,” *Pac J Optim*, vol. 2, no. 1, pp. 35–58, 2006.